Final thesis

Generating web applications containing XSS and CSRF vulnerabilities

by

Gustav Ahlberg

LIU-IDA/LITH-EX-A--14/054--SE

2014-09-30
Final Thesis

Generating web applications containing XSS and CSRF vulnerabilities

by

Gustav Ahlberg

LIU-IDA/LITH-EX-A--14/054--SE

2014-09-30

Supervisor: Ulf Kargén (IDA), Teodor Sommestad (FOI)
Examiner: Nahid Shahmehri
Abstract

Most of the people in the industrial world are using several web applications every day. Many of those web applications contain vulnerabilities that can allow attackers to steal sensitive data from the web application’s users. One way to detect these vulnerabilities is to have a penetration tester examine the web application. A common way to train penetration testers to find vulnerabilities is to challenge them with realistic web applications that contain vulnerabilities. The penetration tester’s assignment is to try to locate and exploit the vulnerabilities in the web application. Training on the same web application twice will not provide any new challenges to the penetration tester, because the penetration tester already knows how to exploit all the vulnerabilities in the web application. Therefore, a vast number of web applications and variants of web applications are needed to train on.

This thesis describes a tool designed and developed to automatically generate vulnerable web applications. First a web application is prepared, so that the tool can generate a vulnerable version of the web application. The tool injects Cross Site Scripting (XSS) and Cross Site Request Forgery (CSRF) vulnerabilities in prepared web applications. Different variations of the same vulnerability can also be injected, so that different methods are needed to exploit the vulnerability depending on the variation. A purpose of the tool is that it should generate web applications which shall be used to train penetration testers, and some of the vulnerabilities the tool can inject, cannot be detected by current free web application vulnerability scanners, and would thus need to be detected by a penetration tester.

To inject the vulnerabilities, the tool uses abstract syntax trees and taint analysis to detect where vulnerabilities can be injected in the prepared web applications.

Tests confirm that web application vulnerability scanners cannot find all the vulnerabilities on the web applications which have been generated by the tool.
Contents

1 Introduction .................................................. 1
  1.1 Motivation ................................................. 1
  1.2 Goals ...................................................... 2
  1.3 Constraints ................................................. 2
  1.4 Outline ..................................................... 2

2 Web Application Attacks ........................................ 4
  2.1 Cross Site Scripting (XSS) attacks ......................... 4
    2.1.1 Reflected ............................................. 5
    2.1.2 Stored ............................................... 7
    2.1.3 Document Object Model (DOM) ....................... 8
    2.1.4 Mutation-based ....................................... 10
  2.2 Cross Site Request Forgery (CSRF) attacks ................. 11

3 Web Application Vulnerability Scanners ......................... 15
  3.1 Detection difficulties ..................................... 16

4 Previous Work .................................................. 18
  4.1 Manually Developed Vulnerable Web Applications ........... 18
  4.2 Generating Vulnerable Web Applications .................... 18
  4.3 Automatically Generating Vulnerable Web Applications ... 20

5 Design and implementation ...................................... 22
  5.1 Phulner .................................................. 22
    5.1.1 Preparing a project .................................. 23
    5.1.2 Generating a vulnerable web application .......... 24
  5.2 Categorization ............................................ 25
    5.2.1 XSS .................................................. 26
    5.2.2 CSRF ................................................ 30
  5.3 XSS Injection .............................................. 30
    5.3.1 Static Taint Analysis ................................ 36
  5.4 CSRF injection ............................................. 37
    5.4.1 Example of a CSRF Vulnerability Injection ......... 38
6 Evaluation
   6.1 Inject exploitable vulnerabilities .................. 42
   6.2 Ability to inject nontrivial vulnerabilities .......... 43
   6.3 Project Preparation Time .......................... 45
   6.4 Future work ....................................... 46

7 Conclusions ........................................ 47
List of Figures

2.1 Steps in a reflected XSS attack ............................. 6
2.2 Steps in a stored XSS attack ............................... 8
2.3 Steps in a DOM XSS attack ................................. 9
2.4 Steps in a Mutation-based XSS attack ...................... 11
2.5 Steps in a CSRF attack ..................................... 13

5.1 Workflow when preparing a project ......................... 23
5.2 Workflow when injecting vulnerabilities .................... 25
5.3 The four dimensions of an XSS vulnerability ............... 26
5.4 Abstract Syntax Tree produced from the code in listings 5.3.1,
5.3.2 and 5.3.3 ............................................. 31
5.5 Abstract syntax tree for the code in Listing 5.3.4 ........ 34
5.6 Abstract syntax tree for the code in Listing 5.3.5 ........ 35
List of listings

2.1.1 Example of a page vulnerable to a reflected XSS attack . . . 7
2.1.2 Example of a page vulnerable to a stored XSS attack . . . . 8
2.1.3 Example of a page vulnerable to DOM XSS . . . . . . . 10
2.1.4 Example of a page vulnerable to Mutation-based XSS . . . . 12
2.2.1 Page vulnerable to CSRF . . . . . . . . . . . . . . . . . . . . 13
4.2.1 Example of Site Generator configuration file . . . . . . . 19
4.3.1 The variable $id$ is sanitized multiple times . . . . . . . . 21
5.1.1 An example project configuration for a Phulner project . . 24
5.1.2 An example instance configuration for a Phulner project . . 25
5.3.1 Code producing the Abstract Syntax Tree in Figure 5.4 . . . 30
5.3.2 Code producing the Abstract Syntax Tree in Figure 5.4 . . . 31
5.3.3 Code producing the Abstract Syntax Tree in Figure 5.4 . . . 32
5.3.4 Code prepared for injection of an XSS vulnerability . . . . . 33
5.3.5 Code for the abstract syntax tree in Figure 5.6 after the san-

5.3.6 Example of how taint propagates. The outputted variable
$number$ is tainted . . . . . . . . . . . . . . . . . . . . . . . . . . 36
5.3.7 Example of how taint propagates and is removed. The out-
putted variable $number$ will not be tainted . . . . . . . . . . . 37
5.4.1 Web page protected from CSRF . . . . . . . . . . . . . . . 40
5.4.2 Web page no longer protected from CSRF . . . . . . . . . . 41
Chapter 1

Introduction

This thesis was conducted at the Swedish Defence Research Agency (FOI) in Linköping, as part of a master’s degree in Computer Science and Engineering at Linköping University.

1.1 Motivation

A vast number of web applications exist today, and most people in the industrialized world are using several of these web applications every day. However, many web application developers lack the skill to write secure code without vulnerabilities [38]. It is therefore likely that some of the web applications contain vulnerabilities. Vulnerabilities in web applications can for example allow attackers to steal sensitive data from the web application’s users [41]. To decrease the number of vulnerabilities in a web application, testing can be done to try to detect, and then fix, the vulnerabilities before the vulnerabilities are exploited. For the tester to be able to detect vulnerabilities in web applications, the testers have to know what to look for in the web applications. Therefore, the testers have to be trained in different methods that can be used to find vulnerabilities in web applications. A common method is penetration testing.

Penetration testing of web applications is done by trying to attack a web application and finding the vulnerabilities in it. One way to train penetration testers is to present a web application to the tester, and inform the tester to find, and exploit, the vulnerabilities in the web application [38]. It is important that the web applications which are used for training is somewhat realistic in terms of functionality and structure, so the vulnerabilities in it are hidden in realistic places. The web applications also have to vary in terms of which vulnerabilities the web applications contain, and how the vulnerabilities can be exploited. Because if the same web application is used twice, the penetration testers who have trained on that web application before already know how to exploit the vulnerabilities in it. Therefore, the
training will not contribute anything new to those penetration testers. For this reason, there needs to be a large repository of different web applications, and variations of the web applications, that contain different kinds of vulnerabilities.

Constructing and maintaining the web applications and their variations manually would be expensive and time consuming. There exist tools which try to solve this problem by trying to inject vulnerabilities into existing code. Some of those tools will be discussed in section 4.2. Those tools either require a lot of preparation work or are not able to automatically inject variations of the same type of vulnerability.

Penetration testing can also be done automatically by using web application vulnerability scanners. Using web application vulnerability scanners is a fast way to find vulnerabilities and can be used in conjunction with manual penetration testing. Penetration testers often use such tools. More about web application vulnerability scanners can be read in Chapter 3.

To know which kinds of vulnerabilities a human penetration tester needs to focus on during a penetration test, it is useful to know which kinds of vulnerabilities a web application vulnerability scanner cannot detect. In Chapter 3, a discussion will be made about which kinds of vulnerabilities are difficult for a web application vulnerability scanner to detect automatically.

1.2 Goals

The goal of this thesis is to create a tool that can inject variations of vulnerabilities in an existing web application. The tool shall be able to inject vulnerabilities which cannot be detected by free current web application vulnerability scanners. This is to ensure that vulnerabilities which need human attention are introduced in the web applications.

1.3 Constraints

- Amongst all possible web application vulnerabilities, only Cross Site Scripting (XSS) and Cross Site Request Forgery (CSRF) will be addressed in this thesis.

- The tool will support injecting vulnerabilities in PHP source files. No other languages will be supported.

1.4 Outline

Chapter 2 explains the web application vulnerabilities that are relevant to this thesis. The next chapter, Chapter 3, explains web application vulnerability scanners. Chapter 4 discusses some previous works which has been conducted in the same area as this thesis. The next chapter, Chapter
5, explains how the tool was designed and how problems were solved. In Chapter 6, the tool is evaluated and it is discussed how the tool can be further developed. Lastly, Chapter 7 contains conclusions drawn from this thesis.
Chapter 2

Web Application Attacks

A web application is an application which runs in a web browser. Usually the source of the web application is hosted on a web server and a web browser makes a request to that web server. The web server responds to the request with a web page. The web page which is sent back is written in browser supported languages, such as HTML, CSS and JavaScript. However, the server could have produced the web page using several other server-side languages. The most common server-side language is PHP, which according to W3Techs is used by over 80% of all the websites that W3Techs has analyzed [40].

For users to be able to communicate with a web application, the web application can receive user input in multiple ways. The most common ways user input is supplied to a web application is in the URL query string (GET parameters), by POST parameters or in cookies [43][18]. A special case of supplying data to a web application is by sending files [31]. The user-supplied data is often used in the web application to create a dynamic web page that is displayed to the user. For example, on a search engine, the user can supply a search query. The search engine will display the search query along with web pages which are related to that search query.

Web applications can contain vulnerabilities and this chapter will describe different types of vulnerabilities and how the vulnerabilities can be exploited. Section 2.1 is about XSS vulnerabilities, and section 2.2 is about CSRF vulnerabilities.

2.1 Cross Site Scripting (XSS) attacks

If a web application contains XSS vulnerabilities, an attacker can inject malicious data onto a web page. The malicious injected data can contain JavaScript which is executed when someone visits the exploited web page. That means an attacker can execute arbitrary JavaScript in a victim’s web browser if the attacker manages to trick the victim into visiting the exploited
2.1. CROSS SITE SCRIPTING (XSS) ATTACKS

CHAPTER 2. WEB APPLICATION ATTACKS

web page. The malicious JavaScript will have the same privileges as any JavaScript supplied by the web application. In other words, the JavaScript will have access to all the data associated with the domain of the vulnerable web application, such as cookies and information in the web application. The result of a successful attack can result in an attacker stealing sensitive data, hijacking a user's session or manipulating the content on the web page [33]. Symantec found in 2012 that XSS was the most common potentially exploitable vulnerability found on websites [19], and CWE/SANS ranks XSS vulnerabilities 4th in their Top 25 Most Dangerous Software Errors [1].

The problem with XSS vulnerabilities is that the web application accepts user-supplied input which later is used as output on a web page without having properly sanitized the user-supplied input. When sanitizing the input, the web application establishes that the input can be safely used as output. This can be done by either checking that the input only contains acceptable values, or manipulating the input. The input can be manipulated in such a way that even if the input contains malicious JavaScript, the JavaScript will not be executed. For example, the input can be encoded, so that when the encoded input is outputted on the web page the browser will treat the output as text, even if the input previously contained HTML tags which would alter the structure of the web page. An example of such an encoding is encoding the special characters in the user's input to HTML entities [24]. Consider the following input:

```html
<script>alert('XSS')</script>
```

the input will, when encoded with the PHP function `htmlspecialchars` and the flag `ENT_QUOTES`, become [16]:

```html
&lt;script&gt;alert('xss');&lt;/script&gt;
```

The encoded representation of the input is safe to use as output in the web application, because the browser would not treat the output as a script tag, but rather as text that will be displayed. Other ways user input can be sanitized is by using a Blacklist or a Whitelist filter [24].

XSS comes in four different flavors: Reflected, Stored, DOM and Mutation-based [33][15]. The different flavors are described in sections 2.1.1-2.1.4 below. It should be noted that some of the examples in this chapter does not work in modern browsers, because of protections such as Reflective XSS Protection, and better client side sanitation [5]. However, the web applications are still susceptible to the XSS attacks depending on which web browser the victim is using.

2.1.1 Reflected

A web application which is susceptible to a reflected XSS attack will use the user-supplied input as output in the web page that is sent as response, without first properly sanitizing the user input [33]. In Figure 2.1, the steps in a reflected XSS attack are shown. The victim supplies input to the web application when requesting a web page. The web application includes the
user-supplied input in the response that is sent back to the victim.

One way an attacker can perform a reflected XSS attack on a web page that have an XSS vulnerability and accepts data in a GET parameter is to craft a URL containing the malicious data. Inside the URL, the attacker can attach the malicious data containing a JavaScript in the GET parameter, and then trick a victim into visiting the crafted URL. When the victim visits the URL, and requests the vulnerable web page with the data the attacker included in the URL, the web application will respond with a web page that includes the malicious data, and the malicious JavaScript in the data will execute.

In Listing 2.1.1, a page vulnerable to a reflected XSS attack is shown. The value of the GET parameter `username` is outputted on the page without any sanitation. If an attacker were to construct the URL: 
```
http://target.com/?username=<script>alert("XSS")</script>
```
and then trick a victim to follow the URL, the value of `username` (`<script>alert("XSS")</script>`) would be supplied to the web application. The web application would respond with a web page including the value of `username`, and the victim’s browser would run the JavaScript in the script-tag. The JavaScript function `alert` would run and display an alert dialog to the victim saying `XSS`. Because the JavaScript is outputted on the page, the browser thinks that this JavaScript was sent intentionally by the web application and therefore, the web browser will execute the JavaScript.
An attacker could execute arbitrary JavaScript instead of displaying an alert dialog.

```php
if (isset($_GET["username"])) {
    echo "Welcome ", $_GET["username"];  
} else {
    echo "<form>";
    echo "<input name='username'>";
    echo "<input type='submit'>";
    echo "</form>";
}
```

Listing 2.1.1: Example of a page vulnerable to a reflected XSS attack

### 2.1.2 Stored

If a web application is susceptible to a stored XSS attack, an attacker can store malicious data which will be used later as output in the web application. The data can be stored anywhere in the application, for example, in a database. The web application contains a page where the stored data is used as output. The stored data is not sanitized before it is stored to the database, nor before it is used as output on a page. When a user requests the page that uses the stored data as output, the data will be retrieved from its stored location and outputted in the response [33]. In Figure 2.2, a stored XSS attack is shown. The attacker supplies data to the web application, and the web application stores the data in a database. When the victim visits the page, the data is retrieved from the database and outputted on the web page that is sent as response.

To understand the possible implications of such a vulnerability think of an online forum. On a forum users can register and choose a user name. The registered user name is saved to the database. The forum will have a page where all the registered users, with their user name, are shown. An example of such a page is shown in Listing 2.1.2. If the user name supplied when registering is not properly sanitized before it is stored in the database, an attacker can choose a user name which will result in JavaScript being executed when it is outputted on the page. For example, the attacker could choose the user name `<script>alert("XSS")</script>`. When the attacker’s user name is later retrieved from the database and outputted on a page, without proper sanitation, the JavaScript will be executed and display an alert dialog to the user that is visiting the page. An attacker could execute arbitrary JavaScript instead of displaying an alert dialog.
2.1. CROSS SITE SCRIPTING (XSS) ATTACKS

CHAPTER 2. WEB APPLICATION ATTACKS

Figure 2.2: Steps in a stored XSS attack

$users = [users from database];
echo "All users: ";
foreach ($users as $user) {
    echo $user["username"], ",";
}

Listing 2.1.2: Example of a page vulnerable to a stored XSS attack

2.1.3 Document Object Model (DOM)

DOM based XSS attacks are similar to reflected XSS attacks. The difference is that no data is sent to the web server, everything is happening in the client’s web browser. This means the web application cannot sanitize the data server-side. Instead, it has to be done on the client-side of the web application [33]. In Figure 2.3, a DOM based XSS attack is shown, and as seen in the figure, the data never leaves the victim’s browser.

In Listing 2.1.3, the GET parameter username is outputted on the page using JavaScript in the client’s web browser. If an attacker constructs the same URL as in the example for the reflected XSS attack in section 2.1.1, http://target.com/?username=<script>alert("XSS")</script> and then trick a victim to follow the URL, the JavaScript on the page would output <script>alert("XSS")</script> without sanitation. The JavaScript would execute and display an alert dialog to the victim. An
Figure 2.3: Steps in a DOM XSS attack
attacker could execute arbitrary JavaScript instead of displaying an alert dialog.

```html
<html><head></head>
<body>
<script type="text/javascript">
  var usernamePos = window.location.search
    .indexOf("username=");
  if (usernamePos >= 0) {
    var username = window.location.search
      .substr(usernamePos + 9);
    document.write(username);
  }
</script>
</body>
</html>
```

Listing 2.1.3: Example of a page vulnerable to DOM XSS

2.1.4 Mutation-based

Mutation-based XSS (mXSS) is an attack which is based on the fact that web browsers automatically tries to fix invalid HTML. The web browser does this by mutating the invalid HTML into valid HTML. Therefore, HTML which seems harmless can mutate into something that executes JavaScript. Because the HTML seems harmless at first, the HTML can bypass many types of sanitation in the web browser and on the server [15]. One thing that triggers the mutation is when using the JavaScript property `innerHTML` to insert new content on a page.

In Figure 2.4, the steps in an mXSS attack are shown. The attacker manages to store seemingly harmless data containing JavaScript in the web application. When the victim visits the web application the seemingly harmless data is outputted to the victim. However, when the victim’s client uses the data as discussed, the victim’s web browser mutates the data and executes the JavaScript that the attacker included in the stored data.

An example of how a web browser (in this case Internet Explorer 8) mutates invalid HTML, is if the input is:

```html
<s class="">hello\&#x20;\<b>world</b></s>
```

Because the HTML is invalid, the browser will mutate the HTML into valid HTML. The result of the mutation is:

```html
<S>hello <B>world</B> </S>
```

The browser has removed the empty class attribute, all the tag names has been converted to upper case, `&amp;#x20;` has been replaced with a whitespace and the `<s>` tag has been closed. This mutation did not result in any executed JavaScript, but in Listing 2.1.4, a page susceptible to an mXSS attack
2.2 CROSS SITE REQUEST FORGERY (CSRF) ATTACKS

A CSRF attack involves three parties, a victim, a trusted website were the victim is currently authenticated and an attacker [34]. In Figure 2.5, an example of a CSRF attack is shown. The victim is currently authenticated on the trusted website, and visits a web page controlled by the attacker. When visiting the attacker controlled web page, the web page tells the victim’s web browser to send a request to the trusted web ap-
2.2. CROSS SITE REQUEST FORGERY (CSRF) ATTACKS

A web page can tell the web browser to send a request to another web application, for example, by inserting the following img-tag:

```
<img src="http://trusted-site.com/change_password.php?
newpassword=pwned">
```

The victim’s web browser will try to fetch the image, i.e. it will send a request to the URL of the image. Because the image is located at the trusted web application’s domain, the victim’s web browser will include all the information the web browser has associated with that domain. This includes the cookies which the web application uses to keep the victim authenticated. Therefore, when the trusted web application receives the request from the victim, the session will be authenticated. To the trusted web application, the request will look like a normal request from the victim’s web browser. In this example, the page which is requested on the trusted web application is changing the victim’s password (Listing 2.2.1 has a copy of the source code for the page). After the victim has made the request to the trusted web application, the victim’s password will have changed to **pwned**.

The problem with CSRF vulnerabilities is that the trusted web application lacks a way to validate that the victim intentionally sent the request. If such validation is missing, an attacker can trick a victim to make an unintentional request to the trusted web application. Because the attacker can make the victim send a request to the trusted web application, a request to make an authenticated action on the victim’s behalf can be made [9]. CSRF

Listing 2.1.4: Example of a page vulnerable to Mutation-based XSS
Figure 2.5: Steps in a CSRF attack

Listing 2.2.1: Page vulnerable to CSRF

```php
if (isset($_SESSION['USERID'])) {
    $currentUser = $_SESSION['USERID'];
    $newPassword = $_GET['newPassword'];
    update_user_password($currentUser, $newPassword);
    echo "Password updated";
}
```

is ranked 12th in CWE/SANS Top 25 Most Dangerous Software Errors \[1\].

A common recommended method that is used to protect against CSRF attacks, is to use a CSRF token \[8][7][34\]. This method is divided into three steps:

**Generate** A token which is hard to guess is generated in the web application on the server. The token is then saved somewhere where the web application can later retrieve it, for example in the client’s cookies.

**Include** The token is sent to the client’s web browser, so that the token will be provided when performing the authenticated action. For example, as a field in a form that will be submitted to perform the authenticated action.

**Guard** When a request to perform an authenticated action is received, the web application will validate the token the client provided, i.e. it shall
be the same as the token the server previously generated. If the tokens do not match, the authenticated action will not be performed.
Chapter 3

Web Application Vulnerability Scanners

A web application vulnerability scanner is a piece of software that is used to automatically scan web applications for vulnerabilities. The web application vulnerability scanner can only send requests and examine the responses from the web applications but it has no knowledge about the source code of the web applications [12]. In other words, the web application vulnerability scanner can only interact with the web application in the same way as an attacker can. Generally speaking, web application vulnerability scanners can be seen as containing three modules: a crawler, an attacker and an analysis module [12]. When initiating a scan, a number of URLs are given to the web application vulnerability scanner. The crawler module will visit the URLs and gather all the URLs on those web pages. Then the crawler module will visit the newly gathered URLs to find even more URLs. This will continue until the crawler module has gathered as many accessible web pages as possible. All input points to the web application (for example form fields and GET parameters) are also gathered while crawling the web application. When the crawler module is finished, it will pass on all the data it has collected (accessible web pages and input points) to the attack module. The attack module constructs attack vectors based on the data from the crawler module. The attack vectors contain input values which are likely to expose a vulnerability in the web application. Based on the attack vectors, the attack module will send requests to the web application. All the responses that are received from the web application are provided to the analysis module. For each response, the analysis module tries to determine if the response contains any traces which indicate the presence of a vulnerability. If the web application vulnerability scanner finds an attack vector which generated a response that indicated the presence of a vulnerability, the web application vulnerability scanner will mark the attacked web page as vulnerable and report which attack vector triggered the vulnerability.
There exist many different web application vulnerability scanners with various features. OWASP and The Web Application Security Consortium both have a list of web application vulnerability scanners [27][37]. From both lists, the free web application vulnerability scanners which could be downloaded at the time of visiting the list were selected. In Table 3.1, the selected application vulnerability scanners are listed, with information regarding whether they support reporting XSS and CSRF vulnerabilities. All of the selected web application vulnerability scanners can be used at no cost without any limitations.

<table>
<thead>
<tr>
<th>Name</th>
<th>Platforms</th>
<th>Last updated</th>
<th>XSS</th>
<th>CSRF</th>
</tr>
</thead>
<tbody>
<tr>
<td>andiparos</td>
<td>OS X, Linux, Windows</td>
<td>2010-10</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Grabber</td>
<td>Python</td>
<td>Not Found</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>OWASP Zed Attack Proxy</td>
<td>Windows, Linux, OS X</td>
<td>2014-05-14</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Paros</td>
<td>Windows</td>
<td>2013-08-14</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Powerfuzzer</td>
<td>Python</td>
<td>2009-01-01</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Skipfish</td>
<td>Linux, OS X, Windows</td>
<td>2012-12</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>w3af</td>
<td>Python</td>
<td>2014-03-31</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Wapiti</td>
<td>Python</td>
<td>2013-10-20</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 3.1: A collection of free web application vulnerability scanners

### 3.1 Detection difficulties

In a test conducted by Bau et al., web application vulnerability scanners performed poorly when the malicious data was first stored in the web application, and then used as output on another page that would trigger the vulnerability [6]. The response from the first request would thus not include anything that would indicate the presence of a vulnerability, because the result of the attack would trigger on another web page in the web application. The same result can also be seen in a test conducted by Doupé et al. [12]. An example of an attack which first stores the malicious data is the stored XSS attack that was discussed in Section 2.1.2.

It is hard for web application vulnerability scanners to detect CSRF vulnerabilities without reporting many false positives (when a vulnerability that cannot be exploited is reported). That is because it is impossible for a web application vulnerability scanner to know which requests need to be protected against CSRF attacks [4]. One way a web application vulnerability scanner could try to detect CSRF vulnerabilities, is to record every request, and when the scan is complete, the web application vulnerability scanner could try to replay every request and report if any of the requests were successful. If a request could be successfully replayed, the scanner could report the web page as possibly containing a CSRF vulnerability. This would, as mentioned previously, result in many false positives, because many of the web pages in a web application do not need to be protected against
CSRF attacks.
Chapter 4

Previous Work

A number of web applications are created to be deliberately vulnerable and used when training and testing web vulnerability scanners. Some of the vulnerable web applications are generated using different tools and others are manually developed.

4.1 Manually Developed Vulnerable Web Applications

There exist web applications that are designed to be vulnerable and to be used when testing web application vulnerability scanners. For example, some companies that offer web application vulnerability scanners also provide a vulnerable web application which can be used to test their web application vulnerability scanner. Examples of such web applications are Altoro Mutual [3], Zero Bank [46] and Acuart [2]. Those vulnerable web applications are useful for testing if a web application vulnerability scanner can detect certain kinds of vulnerabilities. Penetration testing personnel can train by using the vulnerable web applications. However, the vulnerable web application can only be used once for training. As mentioned in Chapter 1.1, if a penetration tester has trained on the web application before then the penetration tester already knows how to exploit all the vulnerabilities and will not learn anything new.

4.2 Generating Vulnerable Web Applications

OWASP\(^1\) had a project called *SiteGenerator*, which allowed the creation of dynamic web applications based on XML files and predefined vulnerabilities [26]. One of the main purposes for the tool was to test and benchmark

\(^1\)The Open Web Application Security Project https://www.owasp.org/
4.2. GENERATING VULNERABLE WEB APPLICATIONS

CHAPTER 4. PREVIOUS WORK

web application vulnerability scanners. The tool consists of two main components: a web server and a GUI application. With an XML file, virtual names are mapped onto files on disk to create the web application. In Listing 4.2.1, an example of a Site Generator XML file is shown. The GUI application loads the XML file and then listens to requests from the web server. When the web server receives a request for the page HelloWorld.aspx from a client, the server asks the GUI application which page to serve. The GUI application responds with the file aspx/Default.aspx to the web server, which serves the file back to the client.

To create a new vulnerability, a new web page has to be created that contain the vulnerability. Then, when creating a dynamic web application, the vulnerable web page could be mapped, using the XML file, to be included in the new web application. The web application will then contain the vulnerability which was included in the web page. This means that for every variant of a vulnerability, a new web page has to be created and added to a web application. Because the vulnerabilities are created manually, it is certain that the web application will be vulnerable when the vulnerability is added. The vulnerabilities which are created can easily be included in any number of web applications. It does, however, require a great deal of work to create all the different variants of the vulnerabilities. The tool makes it easy to create many web applications which consist of different combinations of the same vulnerable web pages. Experts can manually inject vulnerabilities
in web applications but, as stated in the introduction, that would be tedious and it would take a long time to inject many different vulnerabilities. The main benefit of experts manually injecting vulnerabilities is that the expert can create complex vulnerabilities which are dependent on several parts of the web application. In this way, hard and meaningful vulnerabilities can be created.

4.3 Automatically Generating Vulnerable Web Applications

Fonesca et al. proposed a methodology to inject realistic vulnerabilities into existing web applications in [13] and [38]. Based on the methodology, Fonesca et al. implemented a tool which can inject SQL Injection Vulnerabilities into existing web applications. An attacker can execute malicious SQL commands on the database by exploiting a SQL injection vulnerability. This can give the attacker the ability to read, write, delete, insert and execute arbitrary SQL commands on the database [30].

The tool uses Vulnerability Operators that consist of a Location Pattern and a Vulnerability Code Change. The Location Pattern defines the conditions to which a specific vulnerability type must abide. The Vulnerability Code Change specifies how the vulnerability is injected depending on where the vulnerability will be injected [13]. An example of a Vulnerability Operator is the Location Pattern which searches for the PHP function-call `intval`, and where the argument to the function is user input. An example of a code that will match the previously mentioned Location Pattern is:

```php
$id = intval($_GET["id"]);
```

If an SQL query is constructed with the variable `$id` the Vulnerability Code Change consists of removing the intval function-call. After the Vulnerability Code Change is applied, the resulting code will be:

```php
$id = $_GET["id"];  
```

which is vulnerable to an SQL injection attack.

When injecting vulnerabilities, a file is provided as input to the tool. The tool recursively searches for files that are included from the input file. All the included files are concatenated with the input file to create a single file that will be analyzed. The tool searches for all the variables in the file and creates a mesh of how the variables are related to each other based on the names of the variables. Then the tool searches for all the variables which are used to construct a SQL query, and using the previously created mesh, the tool can determine if a variable directly or indirectly influences a SQL query. The last part of the injection stage is to iterate each found variable location and test if any of the Vulnerability Operators are applicable on the location. If an applicable location is found, the Vulnerability Code Change is used to inject a vulnerability at that location. The copy of the file with the injected vulnerability is stored and then the tool enumerates the next
location. When the enumeration is finished, the tool has created a number of files with one possible vulnerability in each.

```php
$id = intval($_GET["id"]);
$query = sprintf("SELECT * FROM users WHERE id = %d", $id);
```

Listing 4.3.1: The variable `$id` is sanitized multiple times

Fonesca et al. identified two situations where the tool failed to inject a vulnerability:

- The first situation is when the same variable name is used both in a SQL query (where it can be used to inject a vulnerability), and in another part of the file where no vulnerability can be injected. When the tool tried to inject vulnerabilities in the second location it did not result in a vulnerability.

- The other situation is when a variable is sanitized multiple times. Because the tool only tries to inject a vulnerability in one location per iteration, it might leave the variable unprotected at the first location. However, before the variable is used in the SQL query the variable is sanitized, and therefore the variable can not be exploited.

An example of such an occurrence is in Listing 4.3.1. The tool will remove the `intval` function, but when the value is inserted in the SQL query with `sprintf` using the `%d` format, `sprintf` will treat the argument as an integer and therefore sanitizes the variable again [35].

The tool also consists of an attack injection part that can test if a vulnerability was injected into the file. If the tool manages to attack the generated file, the tool knows with certainty that the vulnerability was successfully injected.
As stated in the introduction, the aim of this thesis project was to generate web applications which contain vulnerabilities. The generated web application’s purpose is to be used when training penetration testers and evaluating web application vulnerability scanners. A tool shall be designed and implemented to generate these web applications. More specifically, the tool shall be able to:

- Inject XSS and CSRF vulnerabilities in PHP code.
- Be automated as much as possible.
- Inject vulnerabilities that span over more than one file.
- Automatically inject variations of the same vulnerability.
- Inject vulnerabilities that are hard for web vulnerability scanners to detect.
- Be certain that every injected vulnerability is exploitable.

Phulner was created to do that. Section 5.1 describes how Phulner is supposed to be used. Section 5.2 describes the vulnerability categories and dimensions which Phulner can and cannot handle. Sections 5.3 and 5.4 describe the internal workings of Phulner when it handles XSS and CSRF vulnerabilities.
inj.

tion

Figure 5.1: Workflow when preparing a project

5.1.1 Preparing a project

Before Phulner can inject vulnerabilities in a project, the source code must first be prepared. This is because it is hard, on a whole project, to do a static analysis to identify how data flows in and out from the web application. It is particularly hard to statically find the data flow if the input and output are located in different files, and the data might be stored in a database in between input and output. The preparation work limits the amount of source code Phulner has to analyze to inject a vulnerability.

The intended workflow in the preparation of a project can be seen in Figure 5.1. The web application has to first be manually analyzed and prepared. When a location where a vulnerability can be injected is found, keywords have to be placed in the code to tell Phulner to analyze that part of the source code. An example of source code that is prepared to be analyzed by Phulner can be seen in Listing 5.3.4. Between the two keywords /*Phulner*/ on their own lines, a JSON configuration object is found. Then, everything until the last keyword, /*Phulner*/, is the code block that Phulner will analyze. The configuration object includes an identifier, which connects the code block to a specific vulnerability. Every code block with the same identifier is analyzed when the vulnerability identified by the identifier is injected. The configuration object also contains the dimensions (details in section 5.2) in which this part of the vulnerability can vary. Depending on the type of the vulnerability, the object might contain more attributes.

When the source code has been prepared, a configuration file for the whole project has to be created. An example of such a file can be seen in Listing 5.1.1. The file contains a JSON object containing the project’s name, the relative directory where the web application’s source files are placed and a list of all the possible vulnerabilities that can be injected, identified by the identifier. Each vulnerability has a type, a description, a list of files and options. The options object contains different keys depending on the type of the vulnerability. In sections 5.3 and 5.4 the options object for XSS and CSRF vulnerabilities will be discussed.

By first preparing the project, one of the problems Fonesca et al. had
in their tool where the same variable name was used in different locations is avoided (discussed in section 4.3). By limiting the code that Phulner analyzes to parts which are relevant to the vulnerability, the developer can thus be certain that Phulner will inject a vulnerability.

```
{
    "name": "Phulner testproject",
    "basedir": "files/",
    "vulnerabilities": {
        "xss_echoId": {
            "description": "Echoes the user supplied id",
            "type": "xss",
            "options": {
                "input": ["GET"],
                "sanitation": ["NONE", "BLACKLIST"],
                "output": ["NORMAL_TAG"],
                "mutated": [false]
            },
            "files": [
                "welcome.php"
            ],
            "sanitationFunctions": []
        },
        "csrf_changePassword": {
            "description": "Protects the change password action",
            "type": "csrf",
            "options": {
                "types": ["NONE", "ONLY_POST", "COMPUTABLE"]
            },
            "files": [
                "changePassword.php"
            ]
        }
    }
}
```

Listing 5.1.1: An example project configuration for a Phulner project

### 5.1.2 Generating a vulnerable web application

The intended workflow to generate a vulnerable web application from a prepared project is shown in Figure 5.2. First, an instance configuration file has to be created (an example of such a configuration file is shown in Listing
5.1.2). The instance configuration file contains a JSON object specifying from which project Phulner shall generate the vulnerable web application, which vulnerabilities that are to be injected, and in which categories the injected vulnerabilities shall be. When running Phulner, the instance configuration file is provided as input. Phulner will load the specified project and inject the specified vulnerabilities according to the instance configuration file (details on how Phulner injects different vulnerabilities will be discussed in sections 5.3 and 5.4). The newly generated vulnerable web application will be saved at the specified out-path in the instance configuration.

By altering the parameters for the vulnerabilities in the instance configuration file, the same web application, but with other vulnerabilities, can be generated again by running Phulner with the new instance configuration file.

```json
{
    "project": "/path/to/phulner/project/",
    "out": "/destination/of/vulnerable/web/application/",
    "vulnerabilities": {
        "xss_echoId": {
            "inject": true,
            "sanitation": "NONE"
        },
        "csrf_changePassword": {
            "inject": true,
            "type": "NONE"
        }
    }
}
```

Listing 5.1.2: An example instance configuration for a Phulner project

5.2 Categorization

The selected vulnerabilities were divided into different categories in order to distinguish between different variations of the same vulnerability. The categories are also used when detecting which types of vulnerabilities a web application vulnerability scanner can detect. The categories were created based on the characteristics of the vulnerability.
5.2. XSS

When an attacker tries to find an XSS vulnerability, the attacker first tries to detect the way input can be provided to the web application. The next step is to investigate which of the inputs can be used in an attack against the web application. This is done by providing different input and analyzing the response from the server [36]. Based on how an attacker works when trying to detect an XSS vulnerability, we hypothesize that an XSS vulnerability has certain characteristics. The vulnerability starts with something being Inputted to the web application. The input might be partly Sanitized. Then something is Outputted to the browser and the browser might Mutate the output. In Figure 5.3 the characteristics are shown. From these characteristics, we divided an XSS vulnerability into four dimensions.

**Input**

The first thing an attacker has to find out is how to feed input to the application. The input can be supplied in several ways and the input Phulner can handle is categorized in the following way:

- **GET**
  
The input is in a GET parameter.

- **POST**
  
The input is in a POST parameter.

- **Cookies**
  
The input is in a cookie.

- **Header**
  
The input is in a header field.

- **Stored**
  
The input is coming from something that was previously stored in the web application. For example, from a database.
Sanitations

When the ways in which the web application accepts inputs are detected, then the next thing is to detect if the input is sanitized before it is used in the application. Sanitation is used to remove harmful parts from the input and validate the input.

- **None**
  The content is not checked or sanitized in any way.

- **Blacklist**
  A set of characters and sequences are blacklisted and removed from the content.

- **Whitelist**
  A whitelist is the opposite of the Blacklist. Only certain characters and sequences are permitted, and anything else is removed from the content.

- **Encoding**
  A popular method to decrease the risk of attacks when displaying user input is to encode or convert it. If insufficient encoding is applied or not enough is converted, harmful data can still be executed.

Output

Depending on where the content is outputted on the page, the amount of malicious content the attacker has to provide varies [45]. For example, if an attacker were able to inject content into a **Script tag** the content would be executed as JavaScript. In a **Normal tag**, however, the data would not execute as JavaScript. The attacker will have to inject something that makes the client execute the data as JavaScript. The second code snippet in the following examples shows what an attacker could send to be able to run JavaScript in each category. The following list shows the output locations that Phulner can handle.

- **Script tag**
  `<script>HERE</script>`
  The data will be executed as JavaScript.
  `alert("XSS")`

- **HTML Comment**
  `<!-- HERE -->`
  By ending the comment an attacker can insert a script tag.
  `--> <script>alert("XSS")</script> <!--`

- **Attribute name**
<img HERE=x>

Inserting a space would allow the attacker to use any attribute.

onload="alert('XSS')" src

- **Tag name**
  <HERE>
  The attacker can decide which tag will be used. If the attacker also can decide the content of the tag, the attacker could then specify a script tag and anything in the tag would be run. If not, the attacker can add any attribute to the tag.
  
  img onload="alert('XSS')" src="x"

- **Style tag**
  <style>HERE</style>
  In some browsers JavaScript URLs are executed.
  
  body { background:url("javascript:alert('XSS')") }

- **Normal tag**
  <div>HERE</div>
  An attacker can insert a script tag. The content in the script tag is treated as JavaScript, and therefore executed.
  
  <script>alert("XSS")</script>

- **URL attribute - unquoted**
  <a href=HERE>
  If the attacker inserted a JavaScript URL and the user clicked the link, the JavaScript will be executed.
  
  javascript:alert('XSS')"

- **URL attribute - single quotes**
  <a href='HERE'>
  If the attacker inserted a JavaScript URL and the user clicked the link, the JavaScript will be executed.
  
  javascript:alert('XSS')

- **URL attribute - double quotes**
  <a href="HERE">
  If the attacker inserted a JavaScript URL and the user clicked the link, the JavaScript will be executed.
  
  javascript:alert('XSS')

- **Non JavaScript attribute - unquoted**
5.2. CATEGORIZATION

CHAPTER 5. DESIGN AND IMPLEMENTATION

• Non JavaScript attribute - single quotes
  `<input value='HERE'>`
  A single quote would allow the attacker to insert any attribute.
  `x' onclick='alert("XSS")`

• Non JavaScript attribute - double quotes
  `<input value="HERE">`
  A double quote would allow the attacker to insert any attribute.
  `x" onclick="alert('XSS')"

• JavaScript attribute - unquoted
  `<button onclick=HERE>`
  The data will be executed as JavaScript.
  `alert("XSS")`

• JavaScript attribute - single quoted
  `<button onclick='HERE'>`
  The data will be executed as JavaScript.
  `alert('XSS')`

• JavaScript attribute - double quoted
  `<button onclick="HERE">`
  The data will be executed as JavaScript.
  `alert('XSS')`

• JavaScript manipulation
  The content is inserted into the DOM-tree with JavaScript.

• Other location
  Location not in this list.

Mutated

If the content is inserted into the page using, for example, the `innerHTML` property, the content might be mutated as discussed in 2.1.4.

• Yes
  The data is used in a way so that certain clients will mutate the content.

• No
  The data is not used in any way so that any client mutates the content.
5.2.2 CSRF

The categories for CSRF vulnerabilities are based on common mitigation techniques that may not be sufficient to stop a CSRF attack [8].

- **None**
  The server does not perform any checks to ensure that the request was intentionally sent before performing the authenticated action.

- **Only POST requests**
  If the server only accepts POST requests to perform the authenticated actions.
  This will stop attacks which require the victim to click on a link (because that will generate a GET request to the trusted server).

- **Multiple step**
  If the authenticated action requires more than one request to be performed.
  An attacker will have to execute a number of requests in a specific order to successfully execute the attack.

- **Referer header**
  If the only check the server performs is to check the referer header before performing the authenticated action.
  In the referer header the browser can specify from which URI the request originated [17]. The server can block all requests that do not originate from a trusted URI.

- **Computable token**
  As discussed in 2.2, a common way to protect against CSRF attacks is to use a generated token.
  If the supplied token is not random enough, an attacker can compute the token.

5.3 XSS Injection

```python
$foo = intval('bar');
```

Listing 5.3.1: Code producing the Abstract Syntax Tree in Figure 5.4

The tool built by Fonesca et al. (discussed in section 4.3) contained functionality to parse the source code and find specific patterns in the source code. Phulner uses an abstract syntax tree to represent the source code. By using an abstract syntax tree, Phulner can focus on understanding the logic in the code instead of the grammar. The abstract
syntax tree represents the programming language constructs, such as statements, loops and expressions. Therefore, the abstract syntax tree captures the important structure of the source code without the syntactical details such as punctuation [25][21]. For example, the code in listings 5.3.1, 5.3.2 and 5.3.3 will produce the same abstract syntax tree seen in Figure 5.4. The grammar of the code varies between the examples but the logic in the code is the same. The abstract syntax tree also facilitates the analysis of the arguments to a function-call. For example, an XSS vulnerability in the output dimension category **Non JavaScript attribute - single quotes.** The variable that can be exploited is first passed to the PHP function-call `htmlspecialchars` as in the following code:

```php
$src = htmlspecialchars($userinput);
echo "<img src='", $src , "'>";
```

Because by default, `htmlspecialchars` do not encode single quotes, (’), this location can therefore be exploited. If an attacker provided the string: `http://example.com/img.png' onload='alert("XSS")`
then the outputted image tag from the server will be:

```html
<img src='http://example.com/img.png'
    onload='alert('&quot;XSS&quot;')'>
```

which will display an alert dialog saying **XSS** to the user after the image has been loaded. To make `htmlspecialchars` encode sin-
5.3. XSS INJECTION

CHAPTER 5. DESIGN AND IMPLEMENTATION

Listing 5.3.3: Code producing the Abstract Syntax Tree in Figure 5.4

```php
埃foo'} /* a block comment here with the '=' symbol */
= // line comment
intval # another type of comment
/* another block comment including an assignment
 $foo = intval("bar") */ "bar";
```

ingle quotes, the constant `ENT_QUOTES` has to be provided as the second argument [16]. If the variable was passed to `htmlspecialchars` as in the following code:

```php
$src = htmlspecialchars($userinput, ENT_QUOTES);
echo "<img src='", $src , ">";
```

and if the attacker provides the previous string (`http://example.com/img.png' onload='alert("XSS")`), the attack will not be successful. That is because `htmlspecialchars` encodes the single quotes which means that the outputted image tag from the server will be:

```html
<img src='http://example.com/img.png' onload='alert("xss")'></img>
```

Because the single quotes are encoded, the attack can not break out from the src attribute and insert a new attribute that will execute JavaScript. When using an abstract syntax tree, all the information about the function-call, such as the arguments, will be in the function-call’s node. If the tool handled the parsing of the source code, more logic will have to be created to be sure that `ENT_QUOTES` is passed as the second argument to `htmlspecialchars`. Another benefit when using an abstract syntax tree is that, when replacing nodes, it is not possible to break the syntax. A valid abstract syntax tree will always transform into syntactically correct code.

To keep track of how user input is propagated through the web application, Phulner uses a method called Static Taint Analysis. Variables that depend on user input and are not sanitized are labeled as tainted. By using static taint analysis, Phulner can analyze when user input is sanitized in the web application. More details on how static taint analysis works can be found in Section 5.3.1.

To help Phulner understand how functions sanitize variables, a library of sanitation functions is included in Phulner. A sanitation function describes how a specific function works in the aspect of sanitizing its arguments. The sanitation function describes if taint from the arguments to the function is propagated to the return value from the function. When the vulnerability function is used, it is aware about the vulnerability which Phulner is currently trying to inject. Depending on the different options of the vulnerability, such as the output category of the vulnerability, the sanitation function can determine if taint will propagate. The sanitation function for the built-in PHP function `htmlspecialchars` would determine that the re-
5.3. XSS INJECTION

CHAPTER 5. DESIGN AND IMPLEMENTATION

turn value would be tainted if: the first argument is tainted; the constant ENT_QUOTES is not used, and the output category is Non JavaScript attribute - single quotes. However, if the output category was Normal Tag then htmlspecialchars will not propagate the taint from the first argument. In order to be able to execute JavaScript in a normal tag, a new tag such as a script tag has to be added and that is not possible since htmlspecialchars encodes < and > by default. This encoding makes it impossible to create a new tag. The sanitation function also specifies how the function can be replaced to retain the taint from the arguments depending on the options of the vulnerability. How that works will be discussed below.

```php
/*Phulner*/
{
    "identifier": "xss_echoId",
    "sanitation": ["NONE", "BLACKLIST"];
    "initialScope": [
        {
            "name": "userinput",
            "type": "variable",
            "taint": ["USER"]
        }
    ]
}
/*/Phulner*/
$id = intval($userinput);
echo $id;
/*/Phulner*/
```

Listing 5.3.4: Code prepared for injection of an XSS vulnerability.

When injecting an XSS vulnerability, Phulner can vary the vulnerability in the Sanitation dimension. The preparations needed before an XSS vulnerability can be injected consist of: defining the input, output, and mutation category; specifying in which sanitation categories the vulnerability can vary, and specifying which variables are tainted in the beginning of the code block. In Listing 5.3.4, a source code that is prepared for injecting an XSS vulnerability is shown. The code block belongs to the vulnerability xss_echoId, and can be varied in the sanitation dimension to be in the category NONE or BLACKLIST. The variable $userinput is tainted in the beginning of the code block. More information about the vulnerability is found in the project’s configuration file (displayed in Listing 5.1.1). The project’s configuration file specifies that the vulnerability identified by the identifier xss_echoId is of the type XSS. The input is from a GET parameter and the output location will be in a normal tag. The data will not be mutated.

When Phulner processes a code block belonging to an XSS vulnerability
such as the code block in Listing 5.3.4, the code is parsed into an abstract syntax tree using a library called PHP Parser [29]. The resulting abstract syntax tree can be seen in Figure 5.5. An initial variable scope for the code block is constructed from the configuration object. The purpose of the scope is to keep track of which variables contain traces of user input (taint). The abstract syntax tree is then traversed: each node is processed three times. The first processing updates the scope when it encounters a node which changes the scope. If a variable is assigned with something that is tainted without first being sanitized, the taint will propagate to the new variable. The next processing assigns a taint attribute to each node. A node’s taint depends on the type of node which is encountered. For example, if a variable node is encountered, the taint is taken from the current scope. If a function-call node is encountered, Phulner uses the function’s sanitation function to determine if taint propagates from the arguments to the return value of the function. The last processing replaces the nodes that remove taint, which could, for example, be a function-call node. Phulner knows if a function-call removes taint by using the function’s sanitation function, and checking if the arguments to the function are tainted but the return value is not tainted. Phulner can then replace that function so that the taint is retained and a vulnerability has been injected, because user input which has not been sanitized is outputted on the page.

If Phulner was run with the instance configuration file in Listing 5.1.2, it would inject an XSS vulnerability with the sanitation category **None** in the code showed in Listing 5.3.4. First, Phulner will generate the abstract syntax
tree in Figure 5.5. While traversing the tree, Phulner will detect that the node containing the function-call `intval` removed the taint from the variable `$userinput`. During the last processing, the sanitation function for `intval` will be used to look up how `intval` shall be replaced to inject a vulnerability with the sanitation category `None`. In this case, Phulner will replace the `intval` function-call with the argument, and therefore remove the function-call. That replacement will produce the abstract syntax tree in Figure 5.6, which is transformed to the code in Listing 5.3.5. This code is susceptible to an XSS attack because the user input is never sanitized before it is outputted to the user.

Phulner can perform other node replacements to inject other variants of the same vulnerability. Depending on the sanitation category in the instance configuration, Phulner will perform various node replacements to inject other variants of the vulnerability. If a vulnerability with the sanitation category `None` is injected, all attempts to sanitize the variable is removed as in the example above. When the sanitation category is `Blacklist` or `Whitelist`, the replacement is different depending on the output category. The whitelisted/blacklisted characters and sequences would differ depending on where the content is outputted. If the output category is one of the `Non JavaScript attribute` categories, and the quote that surrounds
the attribute is allowed/not blacklisted, then an attacker can supply content
which can break out from that attribute and insert a new attribute that can
execute JavaScript as discussed in section 5.2.1. The whitelisted/blacklisted
characters would allow the surrounding quote to be supplied while disallowing other characters (for example '=' ) to make it harder to exploit the vulnerability. It is the same when the sanitation category is Encoding. As seen in the first example in section 5.3, depending on the output category of the vulnerability, the function htmlspecialchars had to be called with the constant ENT_QUOTES to be safe. If the output category had been Normal tag, using htmlspecialchars without ENT_QUOTES would be sufficient to prevent an attack, because the attacker would not be able to inject something that creates a new tag which could be used to execute JavaScript.

5.3.1 Static Taint Analysis

Taint Analysis is a process of analyzing how tainted data is handled in
an application. The property taint on some data denotes that the data
has originated from a taint source. Taint sources are sources where user-
supplied data are received, for example direct user input or reading a file
that is supplied by a user [23]. In a security context, taint analysis is used
for tracing sensitive or untrusted data. Usually it is of value to analyze
if tainted data can reach a taint sink. A taint sink is a location in the
application that can receive tainted data. Usually, locations that output
data to the user or alter the program’s execution path are considered to be
taint sinks [14]. If tainted data can reach a taint sink, a vulnerability might
be present in the application. For example, a user might be able to make the
application output arbitrary data to another user, or, if taint is considered
to be sensitive data, then that data can be leaked. Data can lose its taint
property if the data is passed through a sanitation function, then the taint
property is removed from the data [32]. After the taint is removed, the data
is considered safe to be used in a taint sink.

```php
$number = $userinput;
echo "Your number is: ", $number;
```

Listing 5.3.6: Example of how taint propagates. The outputted variable $number is tainted

When doing Static Taint Analysis, the process described above is done
by only looking at the application’s source code. In Listing 5.3.6, an exam-
ple source code is shown. The variable $userinput is a direct input from a
user, and therefore originates from a taint source and is marked as tainted.
Performing static taint analysis on the code would report that a vulnera-
bility might be present. The variable $number is assigned to the value of
$userinput which is marked as tainted. On the next line, the same variable
5.4 CSRF INJECTION

Before Phulner can inject a CSRF vulnerability, the source code has to be prepared by defining where the CSRF countermeasures are implemented. It is hard to determine statically if an action is protected by a CSRF countermeasure because there are several possible countermeasures. One common method to protect against CSRF attacks is to use random generated tokens \[8][34][7]. There is no single way that a countermeasure is implemented. It

```php
$number = intval($userinput);
echo "Your number is: ", $number;
```

Listing 5.3.7: Example of how taint propagates and is removed. The outputted variable $number will not be tainted

In Listing 5.3.7 almost the same source code as in the previous example is shown. However, in this version the variable $userinput is passed through the function intval (which returns a integer based on the input [20]) before it is assigned to $number. The function intval is a sanitation function and the taint from $userinput is not propagated to $number. When $number reaches the taint sink, the analysis will not show a possible vulnerability because the data passed through a sanitation function before it reached the taint sink.

Static Taint Analysis has some drawbacks that almost all static analysis processes suffer from. It can sometimes be impossible for the analysis to know all the possible states which the application can have. Consider the code below:

```php
$key = rand();
$var = $array[$key];
```

Because the $key is random, the analysis cannot know what value that variable will have.

The vulnerabilities which are discussed in this thesis try to output arbitrary data to users. Therefore, only taint sinks that output data to the user are relevant, and the only type of taint propagation is direct propagation, because the tool only cares that the data can flow to a taint sink, not how. Direct propagation is when new data is derived from tainted data as in the previous examples [22]. The taint from the user input was directly propagated to the variable $number. The vulnerabilities require the data to be strings. Therefore, all operations which convert strings to, for example integers, are considered to remove the taint from the data.

5.4 CSRF injection

is outputted to the user (the variable is reaching a taint sink). Because the variable has not passed a sanitation function, the variable is still marked as tainted when it reaches the taint sink. When the analysis is completed, it will therefore show that a vulnerability might be present in the application.
is, therefore, hard to develop an algorithm during the short time span of this thesis to correctly identify every part of the countermeasure, and then identify how to remove the countermeasure to inject the vulnerability, and inject different variations of the vulnerability. To make it possible for Phulner to inject a vulnerability, the different sections of the countermeasure (specified in 2.2) have to be defined in the source code. The sections which can be specified are Guard, Generate and Include. The Guard section is where the server validates that the request was sent intentionally, and if not, terminates the execution of the action. In the Generate section, CSRF tokens are generated and stored, for example in the client’s cookies. In the section, Include, the CSRF token is included in the server’s response to the client, in such a way that the client will provide the token in the request to perform the authenticated action. Because the code in the section will be replaced or removed entirely, it is important that the sections only contain code which is related to CSRF protection and not to any other functionality, to prevent breaking the web application when injecting a vulnerability.

5.4.1 Example of a CSRF Vulnerability Injection

In Listing 5.4.1, the page is protected against CSRF with the Double Submit Cookies pattern by including a random token in both the submitted form and a cookie. The tokens are then validated on the server [8]. The code has been prepared to be used by Phulner. In the configuration object for each code block, it is specified that the block is part of the vulnerability with the identifier csrf_changePassword. It is also specified to which section each block belongs.

If Phulner was run with the instance configuration in Listing 5.1.2, Phulner would inject a CSRF vulnerability with the category None. The code Phulner would generate can be seen in Listing 5.4.2. The generated page lacks any means to validate that the request from the user was intentional.

Phulner performs different alternations to the different sections depending on the category of the injected vulnerability. If the category of the injected CSRF vulnerability is None, all of the sections are removed, because then when the server receives a request to perform the authenticated action, the validation is removed and the server will continue to perform the requested action. When the category is Only POST request the Guard section is replaced with code which confirms that the request is a POST request. The other sections are removed. There is no way for Phulner to inject a CSRF vulnerability with the category Multiple step, because that would require splitting the original action into multiple steps. Then, there must be an injection of code that can keep track of at which step the current user is. Then more code must be injected to give the user the ability to progress through the action. With the category Referer header, the guard section is replaced with a check which will validate that the referer header contains a value which is accepted before the server performs the
authenticated action. When Phulner is injecting a CSRF vulnerability with
the category \textbf{Computable token}, the \textbf{Generate} section is replaced with
code that generates the md5 hash of the current date and time on the server
(when looking at the value it may seem that it is random but it is easy
to compute knowing the date and time of the server). This value is then
saved in the client’s cookies as in the example above. The \textbf{Include} section
is replaced so that the generated md5 hash is included in the form that is
posted. Lastly, the \textbf{Guard} section is replaced with code which checks that
the cookie and the submitted value match.
<?php
if (isset($_POST['newpassword'])) {
/*Phulner
{
   "identifier": "csrf_changePassword",
   "action": "guard"
}
/*/Phulner*/
    if ($_POST['csrf_token'] !== $_COOKIE['csrf_token']) {
        die("csrf attempt detected");
    }
/*/Phulner*/
    $newpassword = $_POST['newpassword'];
    $user = get_current_user();
    update_user_password($user, $_POST['newpassword']);
} /*Phulner*/
{  
   "identifier": "csrf_changePassword",
   "action": "generate"
}
/*/Phulner*/
$csrf_token = base64_encode(openssl_random_pseudo_bytes(32));
setcookie("csrf_token", $csrf_token);
/*/Phulner*/
<form method="post">
    New password: <input type="password" name="newpassword">
    <br>
<?php /*Phulner*/
{  
   "identifier": "csrf_changePassword",
   "action": "include"
}
/*/Phulner*/
echo "<input type='hidden' name='csrf_token'>";
echo " value='" . $csrf_token . "'>";
/*/Phulner*/
?>
    <input type="submit" value="Change password">
</form>

Listing 5.4.1: Web page protected from CSRF
<?php
if (isset($_POST['newpassword'])) {
    $newpassword = $_POST['newpassword'];
    $user = get_current_user();

    update_user_password($user, $_POST['newpassword']);
}
?>
<form method="post">
    New password: <input type="password" name="newpassword">
    <br>
</form>

Listing 5.4.2: Web page no longer protected from CSRF
Chapter 6

Evaluation

This chapter presents how Phulner was evaluated according to the requirements presented in Chapter 5. First two projects were prepared that Phulner would use to generate vulnerable web applications. The web applications which Phulner generated from the prepared projects were evaluated using web application vulnerability scanners. To evaluate the time needed to prepare a project to be used by Phulner, a test was conducted where the time it took a person to prepare a project was measured.

6.1 Inject exploitable vulnerabilities

Based on the dimensions of XSS vulnerabilities, there are 680 different variations of that kind of vulnerability. Only a few of the possible XSS vulnerabilities were injected due to the short time span of this thesis. Five different categories for CSRF vulnerabilities exist. However, not all of the categories are possible for Phulner to inject, and therefore only some of them were prepared to be injected.

The popular CMS Wordpress [44] was prepared with the XSS vulnerabilities in Table 6.1, and the CSRF vulnerabilities in Table 6.2. The web forum application phpBB [28] was prepared with the XSS vulnerabilities in Table 6.3.

Using Phulner, the web applications were then generated to include these vulnerabilities. The included vulnerabilities were successfully exploited manually.
6.2 Ability to inject nontrivial vulnerabilities

In this test, the requirements for the web application vulnerability scanners were that they are free to use without any limitations, and the scan is performed automatically. From Table 3.1 in Chapter 3 we chose w3af [39] and Wapiti [42]. Due to the time span of this thesis, only two scanners were used to evaluate the web applications that Phulner generated.

Before a scan was performed, the web applications were restored to a baseline, and the database was emptied to remove data that might persist from a previous scan. This was done so that both scanners will scan on the same terms. The vulnerability was first exploited manually to confirm that the vulnerability was present in the web application and possible to exploit. The web application vulnerability scanners were not configured before scanning the web applications, because the web application vulnerability scanners were seen as tools which required no additional time to use.

The results from running both the scanners on the different injected vulnerabilities are shown in tables 6.4 and 6.5. Both scanners found the

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
<th>Sanitation</th>
<th>Mutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>Non JavaScript attribute - double quotes</td>
<td>None</td>
<td>No</td>
</tr>
<tr>
<td>GET</td>
<td>Non JavaScript attribute - double quotes</td>
<td>Blacklist</td>
<td>No</td>
</tr>
<tr>
<td>Stored</td>
<td>Normal tag</td>
<td>None</td>
<td>No</td>
</tr>
<tr>
<td>Stored</td>
<td>Normal tag</td>
<td>Blacklist</td>
<td>No</td>
</tr>
<tr>
<td>Stored</td>
<td>URL attribute - single quoted</td>
<td>None</td>
<td>No</td>
</tr>
<tr>
<td>Stored</td>
<td>URL attribute - single quoted</td>
<td>Blacklist</td>
<td>No</td>
</tr>
<tr>
<td>Stored</td>
<td>URL attribute - single quoted</td>
<td>Whitelist</td>
<td>No</td>
</tr>
<tr>
<td>Stored</td>
<td>URL attribute - single quoted</td>
<td>Encoding</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 6.1: The XSS vulnerabilities that were injected in Wordpress

<table>
<thead>
<tr>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
</tr>
<tr>
<td>Only POST request</td>
</tr>
<tr>
<td>Computable token</td>
</tr>
</tbody>
</table>

Table 6.2: The CSRF vulnerabilities that were injected in Wordpress

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
<th>Sanitation</th>
<th>Mutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST</td>
<td>Normal tag</td>
<td>None</td>
<td>No</td>
</tr>
<tr>
<td>POST</td>
<td>Non JavaScript attribute - double quotes</td>
<td>None</td>
<td>No</td>
</tr>
<tr>
<td>POST</td>
<td>Non JavaScript attribute - double quotes</td>
<td>Blacklist</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 6.3: The vulnerabilities that were injected in phpBB

6.2 Ability to inject nontrivial vulnerabilities
6.2. ABILITY TO INJECT NONTRIVIAL VULNERABILITIES

Most trivial vulnerability where the input came from a GET parameter and was directly outputted on the page without any sanitation. When a blacklist was applied on the input, only one of the scanners was still able to exploit the vulnerability. As discussed before in Section 3.1, it is difficult for scanners to detect vulnerabilities where the input is stored before it is outputted on the web page. This can also be seen in this test, since none of the two web application vulnerability scanners detected the vulnerabilities where the input was first stored and then later outputted on the page. Another reason why the vulnerabilities might be hard to detect is that some forms have to be filled in with acceptable values before the web application even begins to process the form submission, for example, if there is a form to reply to a forum post and the reply content is vulnerable to an XSS attack. If the form contains a field where a valid email address has to be provided, the web application vulnerability scanner has to be able to provide a valid email address to even have the form submission accepted by the server. Because the scanner is not configured in any way, the scanner has to work out by itself that it has to supply a valid email address in that specific field. If the scanner cannot do that, the scanner will not be able to find the vulnerability. Another problem when scanning is that the web application can be large and contain a vast amount of different pages. The web application vulnerability scanner has to crawl every page to find all the inputs where a vulnerability might be present. The task to find all possible pages might be too expensive for the scanner to perform. Therefore, the web application vulnerability scanner might miss pages which could have contained vulnerabilities.

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
<th>Sanitation</th>
<th>Mutation</th>
<th>w3af</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>Non JavaScript attribute - double quotes</td>
<td>None</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>GET</td>
<td>Non JavaScript attribute - double quotes</td>
<td>Blacklist</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>POST</td>
<td>Normal tag</td>
<td>None</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Stored</td>
<td>Normal tag</td>
<td>None</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Stored</td>
<td>Normal tag</td>
<td>Blacklist</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Stored</td>
<td>URL attribute - single quoted</td>
<td>None</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Stored</td>
<td>URL attribute - single quoted</td>
<td>Blacklist</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Stored</td>
<td>URL attribute - single quoted</td>
<td>Whitelist</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Stored</td>
<td>URL attribute - single quoted</td>
<td>Encoding</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 6.4: The results when scanning the generated web applications for XSS vulnerabilities with w3af

Only w3af had the support to scan for CSRF vulnerabilities and the result of the scans can be seen in Table 6.6. Although w3af found a possible CSRF vulnerability on two of the categories, w3af also reported many false positives. w3af reported a possible CSRF vulnerability for every page that included either a POST or a GET parameter unless one of the parameters had a name which looked like a CSRF token or if the value of the parameter looked as if it could be a CSRF token. Because the category Computes
6.3 Project Preparation Time

To evaluate the time needed to prepare a project for someone who has not worked with Phulner before a test was conducted with computer security expert Jonas Almroth who is working at FOI. Jonas Almroth was asked to try to prepare a project with a new vulnerability. If the time it took for me to prepare a project with a new vulnerability, was measured, the result would not be representative of the average user, because I already know exactly how the tool works.

First, a short introduction to Phulner was given and an example of a prepared vulnerability was shown. Then I showed the subject the web application and where I wanted him to prepare a new vulnerability. The subject was not familiar with the project’s source code before this test was performed.

Table 6.5: The results when scanning the generated web applications for XSS vulnerabilities with Wapiti

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
<th>Sanitation</th>
<th>Mutation</th>
<th>Wapiti</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>Non JavaScript attribute - double quotes</td>
<td>None</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>GET</td>
<td>Non JavaScript attribute - double quotes</td>
<td>Blacklist</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>POST</td>
<td>Normal tag</td>
<td>None</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Stored</td>
<td>Normal tag</td>
<td>None</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Stored</td>
<td>Normal tag</td>
<td>Blacklist</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Stored</td>
<td>URL attribute - single quoted</td>
<td>None</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Stored</td>
<td>URL attribute - single quoted</td>
<td>Blacklist</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Stored</td>
<td>URL attribute - single quoted</td>
<td>Whitelist</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Stored</td>
<td>URL attribute - single quoted</td>
<td>Encoding</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 6.6: The results when scanning the generated web applications for CSRF vulnerabilities with w3af

<table>
<thead>
<tr>
<th>Type</th>
<th>w3af</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Yes</td>
</tr>
<tr>
<td>Only POST request</td>
<td>Yes</td>
</tr>
<tr>
<td>Computable token</td>
<td>No</td>
</tr>
</tbody>
</table>

Token adds a hidden input called phulner.csrf.token with a md5 as value, w3af picked it up as a CSRF token and did not report it as a possible CSRF vulnerability. This result confirms what was discussed in section 3.1. It is difficult for web application vulnerability scanners to detect CSRF vulnerabilities without many false positives. Because the way w3af reported possible CSRF vulnerabilities the information is not reliable. A human still has to confirm that the vulnerabilities exist. The report only summarizes where CSRF vulnerabilities can be present.

6.3 Project Preparation Time

To evaluate the time needed to prepare a project for someone who has not worked with Phulner before a test was conducted with computer security expert Jonas Almroth who is working at FOI. Jonas Almroth was asked to try to prepare a project with a new vulnerability. If the time it took for me to prepare a project with a new vulnerability, was measured, the result would not be representative of the average user, because I already know exactly how the tool works.

First, a short introduction to Phulner was given and an example of a prepared vulnerability was shown. Then I showed the subject the web application and where I wanted him to prepare a new vulnerability. The subject was not familiar with the project’s source code before this test was performed.
conducted. Within 10 minutes, the subject had located the relevant source code and inserted the Phulner keywords. The project could now be used by Phulner to inject the new vulnerability. The subject spent most of the time trying to figure out how the project worked and finding the relevant source code. Because the needed sanitation functions were already implemented in Phulner, the subject did not need to spend any time in defining those sanitation functions. That is supposed to be the normal use case when preparing a project that is used by Phulner.

This test shows that without any prior knowledge of Phulner or the source code, it requires little effort to prepare a project to include another vulnerability which can be injected by Phulner.

6.4 Future work

All tools can always be improved and extended to support even more features. Some concrete improvements that could be made to Phulner are:

- Implement support to inject other types than XSS and CSRF vulnerabilities. The implementation to add support for SQL Injections [11] and Command Injection [10] vulnerabilities would be similar to the currently implemented functionality to inject XSS vulnerabilities.

- Implement sanitation functions for more built-in PHP function-calls. Currently, only a handful of sanitation functions are included in Phulner.

- When injecting XSS vulnerabilities, taint analysis is being used. Currently the taint analysis can only handle tainted arrays and variables. Support for objects can be implemented to make Phulner able to handle more source code.

In this thesis, only free scanners were used to evaluate the generated web applications. It would be interesting to investigate if enterprise scanners would do a better job at detecting the vulnerabilities that were injected by Phulner. It would also be interesting to have experienced penetration testers try to exploit the generated web applications.
Chapter 7

Conclusions

In order to find vulnerabilities in web applications, human attention is still needed. As seen in the evaluation of two web vulnerability scanners in Section 6.2, not much was found when the scanners were running without any human interaction. The vulnerabilities that were especially difficult for the scanners to detect were the vulnerabilities where data was stored before it was outputted to the page. Because those vulnerabilities affect all the users that visit the vulnerable web application and not just those who are tricked to visiting a specific URL from an attacker, those vulnerabilities are the most crucial to find. The same case applies to CSRF vulnerabilities, because even if the web application vulnerability scanners find the vulnerabilities, the web application vulnerability scanners report even more false positives. Therefore, human attention is still necessary to confirm the vulnerability. All of this leads to the conclusion that it is still important to train human penetration testers to find vulnerabilities in web applications and not just relying on tools.

To train the penetration testers, Phulner can be used to reliably generate web applications that include vulnerabilities. When a project is prepared to work with Phulner, changing a few parameters can generate different versions of the same web application. Therefore, when the penetration tester is presented with a web application which has been generated by Phulner, the penetration tester cannot be sure exactly which version of the web application it is. Even if the penetration tester has been faced with the same web application before, there will be different vulnerabilities and other variants of the vulnerabilities. From the tests done in this thesis we can draw the conclusion that Phchner can inject vulnerabilities which are not possible for web application vulnerability scanners to detect. Some of the injected vulnerabilities need human attention to be found.

To prepare a project to be used by Phulner only a small amount of time is needed, even if the person who is doing the preparations has no prior knowledge about the project that is being prepared. Most of the preparation
time is spent on finding the relevant source code, and understanding the inner workings of the project and not the specific Phulner parts.

The biggest advantages of using Phulner compared to using the other tools which automatically generate vulnerable web applications, is that Phulner can inject several variations of the same vulnerability without having to write every specific variation of the vulnerability. The vulnerabilities can also be more advanced and span more than one file in the web application. When Phulner has been executed, it is sure that every vulnerability Phulner was configured to inject is exploitable.
Bibliography


På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Uphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Gustav Ahlberg